

Secrets behind PingPongRoot

Wen Xu @ Keen Team

HITB GSEC 2015



PINGPONG ROOT

KEEN TEAM

PingPongRoot

- Credit: wushi, Memeda Xu, James Fang and Leo.C
- First known 64bit root case in the world
- Universally applied bug and exploitation techniques in kernel
- Nominee for Pwnie Awards 2015



Mario Tomás Serrafero · May 9, 2015 at 11:00 am

XDA Picks: Best Apps of the Week (May 1 – 8)

Apps are at the front and center of any smartphone experience, and with over a million apps on the Google Play Store and new apps being submitted to our forums every day, staying up to date on the latest apps and games can be a hassle. At XDA we don't discriminate apps – if it's interesting, innovative, original or useful, we mention them. The XDA Portal Team loves apps too, and here are our top picks for this week.

PingPongRoot – Bypass KNOX! [GALAXY S6 ONLY]



Some people are afraid to root devices due to warranty concerns – at XDA are a little braver than that. However, Samsung phones come with the infamous KNOX trigger (and its mythical hardware fuse) that makes the whole process more of a headache. Every now and then, root methods that don't involve ODIN nor flashing KNOX-tripping software/firmware pop up – and when they do, **they get a lot of love**. PingPongRoot is the latest in S6 rooting, and if you want your warranty safe, this is where you should go. Make sure to thank the developers in whichever way you can, things like these are Samsung phones' Holy Grails!

AIDA64 – Know All About Your Phone [FREE]



AIDA64 for Windows has been a favorite for many PC power users, and now the information utility comes to those of Android. Sure, there are many apps to get similar information (CPU Z is a personal favorite), but this is one of the most data-packed ones out there, with plenty of facts and numbers about your phone and real-time information displays of important hardware and software bits. If you are a power user who loves to test tweaks, needs poll sensors, or wants to remember system properties, this is the app for you. Oh, and it has an native app for Wear Watches too. These are great tools for power users, so keep one handy at all times!

ChronoSnap – Pictures Every Now and Then [FREE]



There are applications to automatically take pictures between intervals, but mostly do so off video and the resulting images don't have the full resolution, or sometimes have terrible video compression artifacts. This is where ChronoSnap comes in: it is an open source application that allows you to set up the intervals, capture or time limits and run the app with the screen off. The app also has a simple and efficient material design, and the developer will continue to add features such as RAW capturing, changing camera parameters and a persistent focus lock. If you are runny a bare-bones stock camera, this is a great solution to a common photography need.

weibo.com/1400697187

Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future

CVE-2015-3636

- Crash log from Trinity
- A critical paging fault at **0x200200**

```
2301 [ 3354.778717] Unable to handle kernel paging request at virtual address 00200200
2302 [ 3354.778839] pgd = ea574000
2303 [ 3354.778900] [00200200] *pgd=00000000
2304 [ 3354.779052] Internal error: Oops: 805 [#1] PREEMPT SMP ARM
2305 [ 3354.779144] Modules linked in:
2306 [ 3354.779266] CPU: 1 Tainted: G W (3.4.0-Kali-g006dd6c #1)
2307 [ 3354.779357] PC is at ping_unhash+0x50/0xd4
2308 [ 3354.779479] LR is at _raw_write_lock_bh+0xc/0x8c
2309 [ 3354.779541] pc : [<c08b18b>] lr : [<c09f7d9>] psr: 20010013
2310 [ 3354.779541] sp : e99a5ee0 ip : c08a67ac fp : 00000000
2311 [ 3354.779724] r10: 00000000 r9 : e99a4000 r8 : c000e928
2312 [ 3354.779846] r7 : 0000011b r6 : 00000053 r5 : 00000000 r4 : eb3cd200
2313 [ 3354.779907] r3 : 00000003 r2 : 00200200 r1 : 00000000 r0 : c144ed98
2314 [ 3354.780029] Flags: nzCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user
2315 [ 3354.780120] Control: 10c5787d Table: ab97406a DAC: 00000015
```



```
34 void ping_unhash(struct sock *sk)
35 {
36     struct inet_sock *isk = inet_sk(sk);
37     pr_debug("ping_unhash(isk=%p,isk->num=%u)\n",
38             |isk, isk->inet_num);
39     if (sk_hashed(sk)) {
40         write_lock_bh(&ping_table.lock);
41         hlist_nulls_del(&sk->sk_nulls_node);
42         sock_put(sk);
43         isk->inet_num = 0;
44         isk->inet_sport = 0;
45         sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
46         write_unlock_bh(&ping_table.lock);
47     }
48 }
49 EXPORT_SYMBOL_GPL(ping_unhash);
```

struct sock sk:
PING socket object in kernel

Allocation: user_space_fd =
socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);


```

71 static inline void __hlist_nulls_del(struct hlist_nulls_node *n)
72 {
73     struct hlist_nulls_node *next = n->next;
74     struct hlist_nulls_node **pprev = n->pprev; 2
75     *pprev = next; 3
76     if (!is_a_nulls(next))
77         next->pprev = pprev;
78 }
79
80 static inline void hlist_nulls_del(struct hlist_nulls_node *n)
81 {
82     __hlist_nulls_del(n);
83     n->pprev = LIST_POISON2; 1
84 }

```


LIST_POISON2 == 0X2000200

ping_unhash(hlist_nulls_del)  two times
ox200200 not mapped

Kernel crash due to
a paging fault


```
1 int inet_dgram_connect(struct socket *sock,
2                        struct sockaddr * uaddr,
3                        int addr_len, int flags)
4 {
5     struct sock *sk = sock->sk;
6
7     if (addr_len < sizeof(uaddr->sa_family))
8         return -EINVAL;
9     if (uaddr->sa_family == AF_UNSPEC)
10        return sk->sk_prot->disconnect(sk, flags);
11
12     if (!inet_sk(sk)->inet_num && inet_autobind(sk))
13         return -EAGAIN;
14     [...]
15 }
16 EXPORT_SYMBOL(inet_dgram_connect);
```

```
19 int udp_disconnect(struct sock *sk, int flags)
20 {
21     struct inet_sock *inet = inet_sk(sk);
22
23     sk->sk_state = TCP_CLOSE;
24     [...]
25     if (!(sk->sk_userlocks & SOCK_BINDPORT_LOCK)) {
26         sk->sk_prot->unhash(sk);
27         inet->inet_sport = 0;
28     }
29     sk_dst_reset(sk);
30     return 0;
31 }
32 EXPORT_SYMBOL(udp_disconnect);
```



Road to ping_unhash()

In user space: **connect()** on socket fd in user space with AF_UNSPEC

In kernel: **udp_disconnect()** on sk (kernel sock object)

Review

ping_unhash()

- First, to avoid crash: map **0x200200** in user space
- `sock_put(sk)` could then be invoked twice.

```

34 void ping_unhash(struct sock *sk)
35 {
36     struct inet_sock *isk = inet_sk(sk);
37     pr_debug("ping_unhash(isk=%p, isk->num=%u)\n",
38             isk, isk->inet_num);
39     if (sk_hashed(sk)) {
40         write_lock_bh(&ping_table.lock);
41         hlist_nulls_del(&sk->sk_nulls_node);
42         sock_put(sk);
43         isk->inet_num = 0;
44         isk->inet_sport = 0;
45         sock_prot_inuse_add(sock_net(sk), sk->sk_prot, -1);
46         write_unlock_bh(&ping_table.lock);
47     }
48 }
49 EXPORT_SYMBOL_GPL(ping_unhash);

```


USE-AFTER-FREE

- ◉ 1. Allocate ping socket and get a file descriptor fd.
- ◉ 2. Map 0x200200 in user space.
- ◉ 3. Connect() to fd with sa_family == AF_INET
 - ◉ To make sock sk hashed in kernel
- ◉ 4. Connect() to fd with sa_family == AF_UNSPEC twice
 - ◉ sock_put(sk) is invoked twice
 - ◉ ref_count of sk is to be 0 → sk is freed in kernel space
- ◉ 5. We have a "dangling" file descriptor fd in our hand now.

```

52  static inline void sock_put(struct sock *sk)
53  {
54      if (atomic_dec_and_test(&sk->sk_refcnt))
55          sk_free(sk);
56  }
    
```


Exploitable?

- ◉ If the space for the freed sk is re-controlled by us,
 - ◉ In user space: simply close(fd)
 - ◉ In kernel:
 - ◉ inet_release() invoked
 - ◉ As sk_prot is controlled by us, we can hijack the control flow of the kernel.
 - ◉ Return to shellcode in usersland or ROP

```

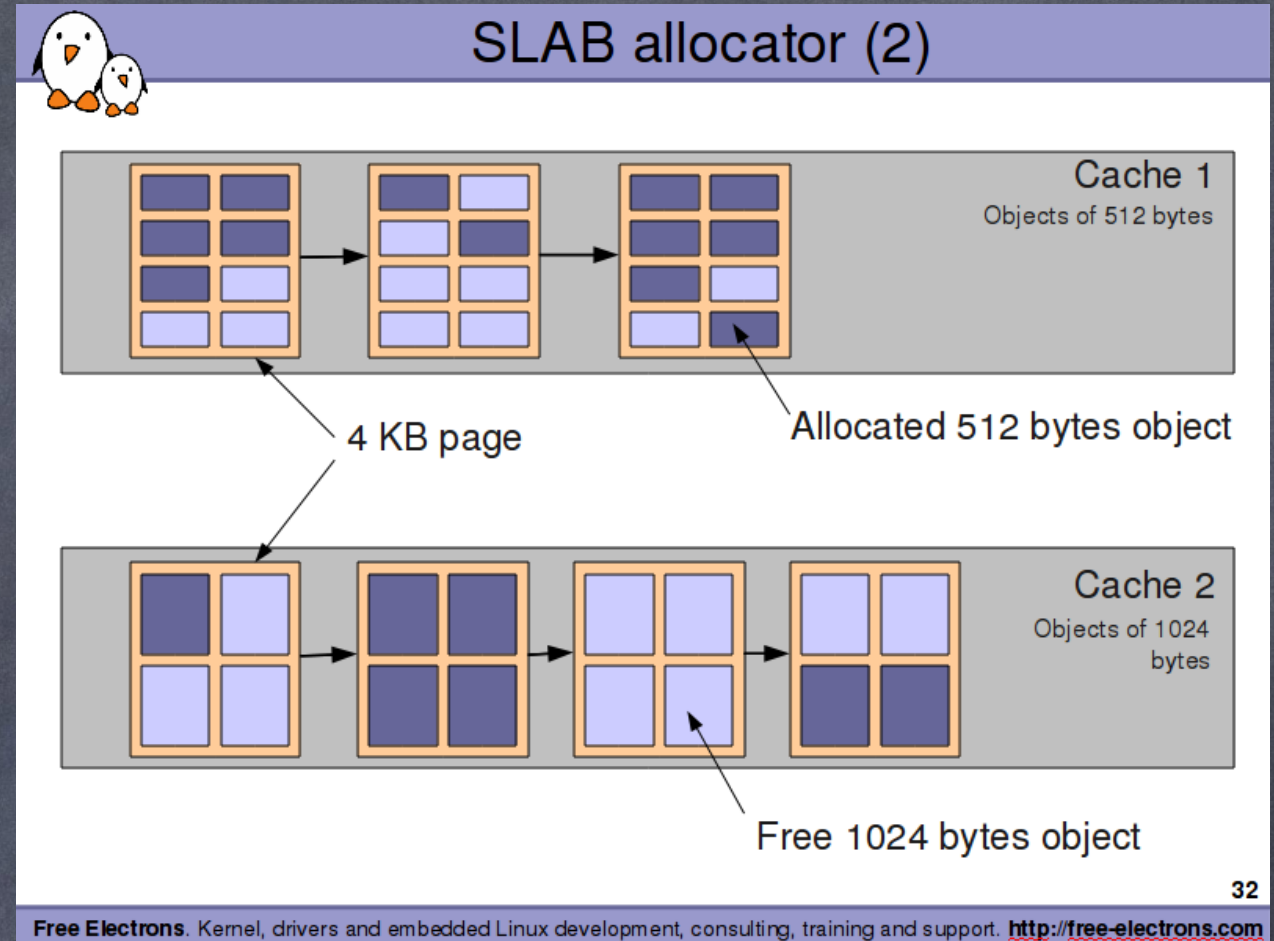
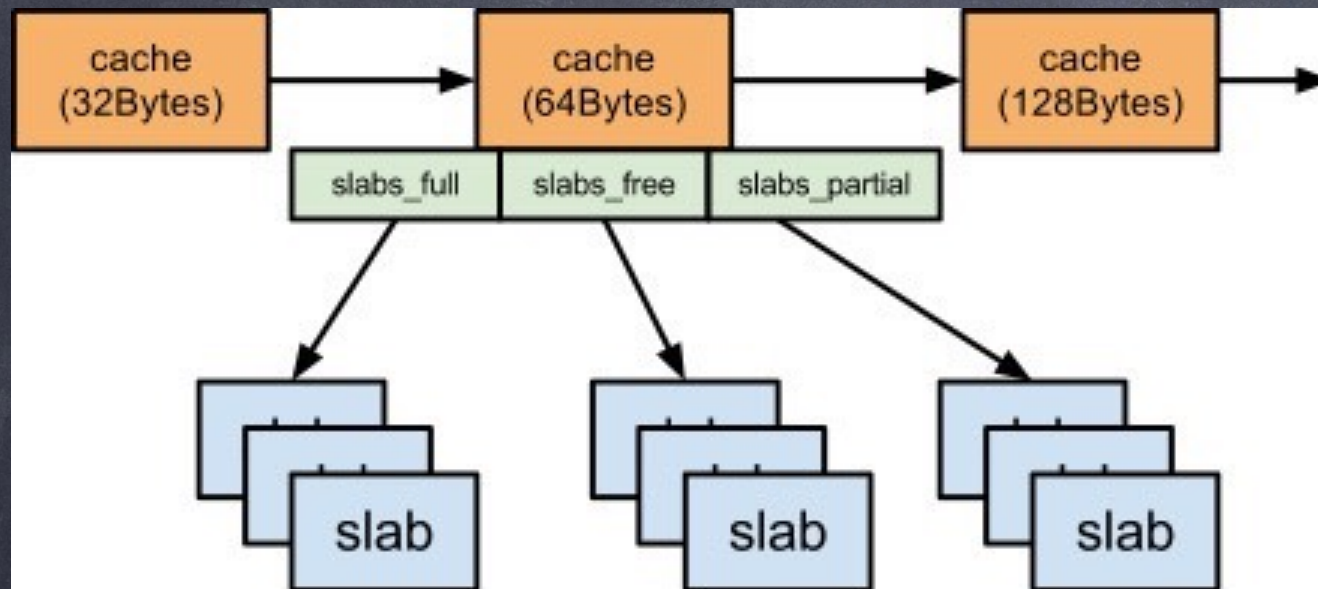
97  int inet_release(struct socket *sock)
98  {
99      struct sock *sk = sock->sk;
100
101      if (sk) {
102          long timeout;
103
104          [...]
105
106          if (sock_flag(sk, SOCK_LINGER) &&
107              !(current->flags & PF_EXITING))
108              timeout = sk->sk_lingertime;
109          sock->sk = NULL;
110          sk->sk_prot->close(sk, timeout);
111      }
112      return 0;
113  }
114  EXPORT_SYMBOL(inet_release);
    
```


Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future

When it comes to UAF

- Most critical step: re-filling the freed vulnerable object
- This time, our target is struct sock object
- And it belongs to cache "PING",
 - `kmem_cache_alloc("PING", priority &__GFP_ZERO);`
 - A custom-use cache



32

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>

SLAB CACHE

A specific area for the allocation of kernel objects
of particular type
Here we meet the type called "PING"

Challenges

- ① 1. Slab allocator
 - ① Natural separation between kernel objects
- ① 2. Few Candidate kernel objects
 - ① Most are not directly under the control of us
- ① 3. Multi-thread/core
 - ① Hard to achieve predictable memory layout
- ① 4. Controllable content
 - ① The content of most kernel objects are not totally under the control of us

What used to Re-Fill?

Candidate #1: `kmalloc()` buffers

- General use SLAB cache
 - Rounded sizes
 - 32, 48, 64, 128, 256, 512, 1024...
- Easy to create: syscall `sendmmsg()`
- Size control: length of control message
- Content control: content of control message

Intuitive Idea

- ◉ Basically, a completely free slab has large probability to be recycled for future allocation
 - ◉ The fact kernel resumes memory provides us the opportunity to exploit UAF bugs anywhere.
- ◉ 1. Fill slabs with totally PING socket objects
- ◉ 2. Free all of them and spray kmalloc-x buffers
- ◉ Exactly possible, but ... **out of control**
 - ◉ Low success rate in practical

SLUB Help Us?

- ◉ Newly adopted SLUB allocator tries to put **the objects of the same size** together, which de-separates the kernel objects to some extent
- ◉ Then, does our target object have a size of 32, 48, 64, 128, 256 or 512?

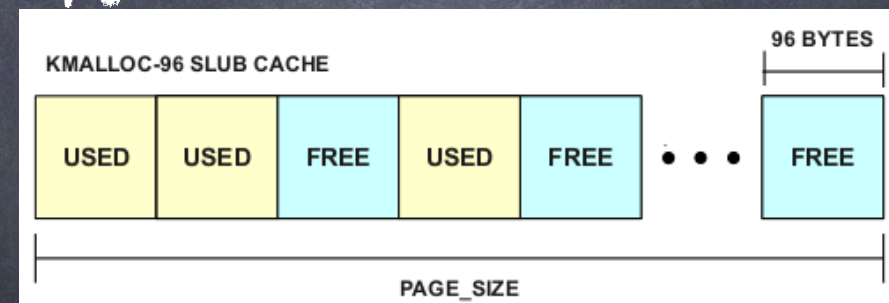
- ◉ Use `kmalloc()` buffers to re-occupy

- ◉ Much more stable and accurate

- ◉ Limitations:

- ◉ What if the size of the vulnerable object is 576, where $512 < 576 < 1024$?

- ◉ Sizes of PING sock objects varies on different devices

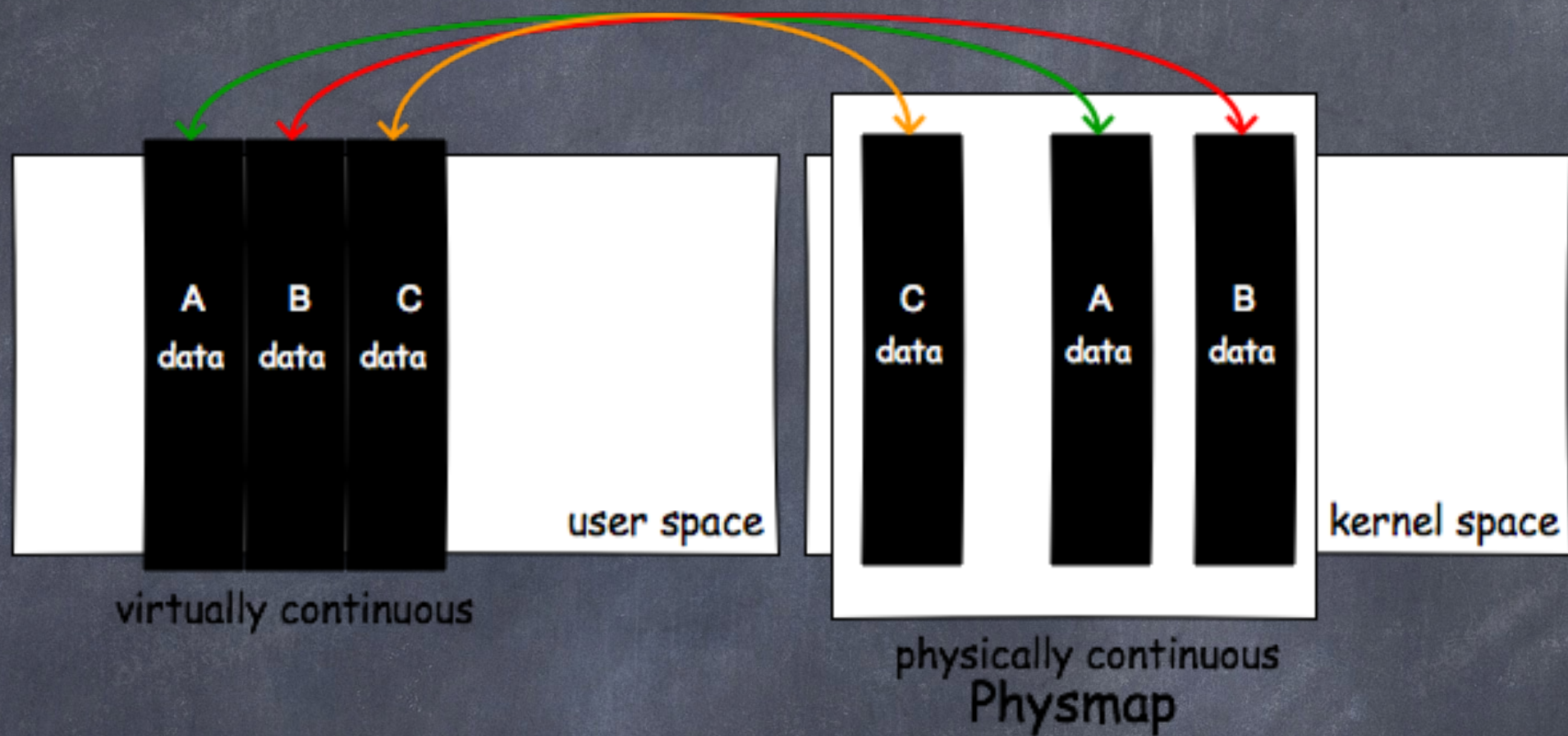


Memory Re-filling

Universal Solution #1

RET2DIR

- ret2dir: Rethinking Kernel Isolation (USENIX 14')
 - Vasileios P. Kemerlis Michalis Polychronakis
Angelos D. Keromytis
- **Physmap** is supposed to bypass kernel protections in the paper
 - SMEP, SMAP, PXN, PEN ...
- Would it help exploit kernel use-after-free bugs as well?



The Return of Physmap

Physmap, the direct-mapped memory, is memory in the kernel which would directly map the memory in the user space into the kernel space.

The Return of Physmap

- Easy to create: iteratively calling `mmap()` in user space
 - `mmap((void *)addr, 0x100000000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_SHARED | MAP_FIXED | MAP_ANONYMOUS, -1, 0);`
- Data control: fully user-controlled (fill `mmap()`'ed area with our payload)
- Physmap with payload grows by occupying the free memory in the kernel

The Return of the Physmap

- Size control:

- Physmap does not care about the size(type) of the vulnerable object

- For its self, it has a large effective range.

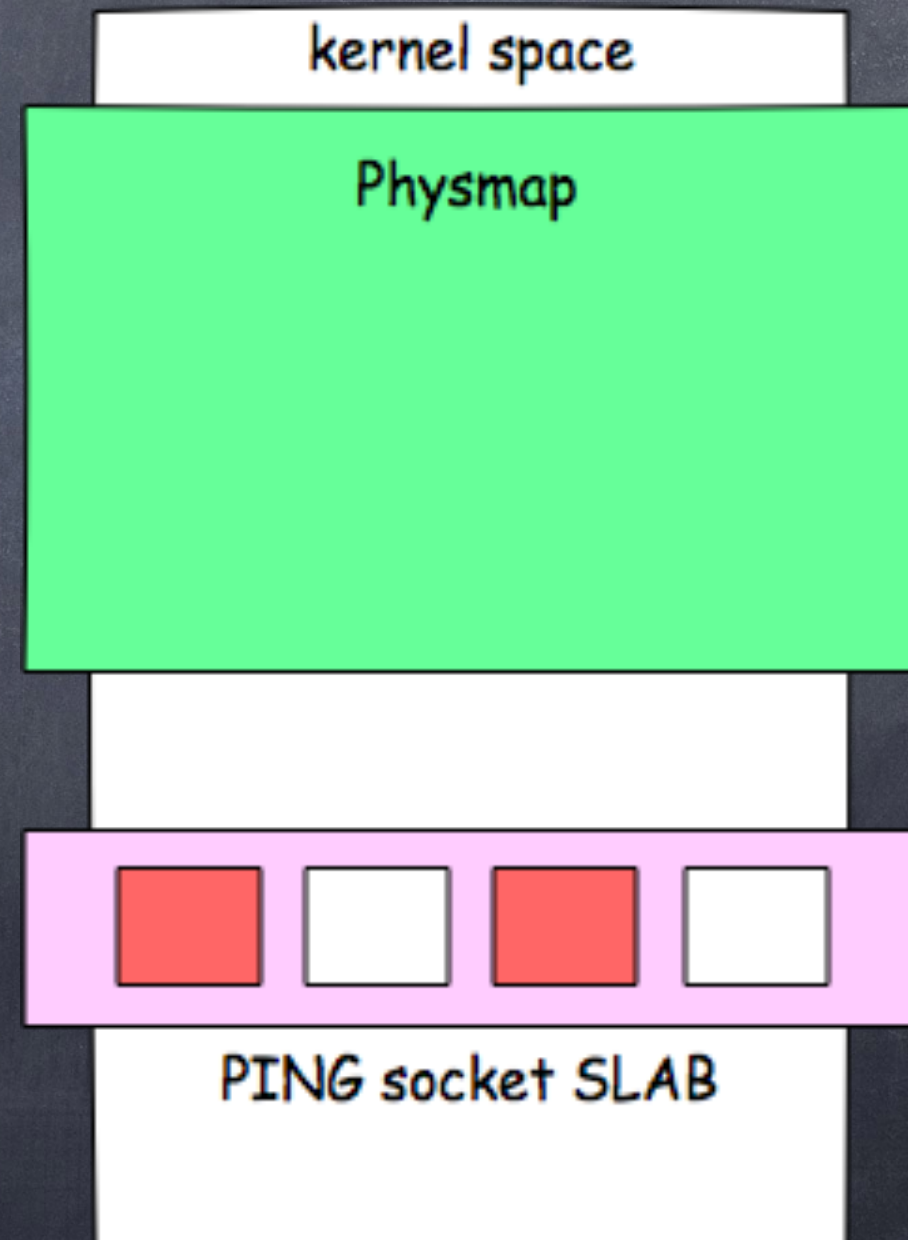
- Large enough to fill any pages of freed memory theoretically

Architecture	PHYS_OFFSET	Size	Prot.
x86	(3G/1G)	0xC0000000	891MB
	(2G/2G)	0x80000000	1915MB
	(1G/3G)	0x40000000	2939MB
AArch32	(3G/1G)	0xC0000000	760MB
	(2G/2G)	0x80000000	1784MB
	(1G/3G)	0x40000000	2808MB
x86-64	0xFFFFF88000000000	64TB	RW (X)
AArch64	0xFFFFF00000000000	256GB	RW (X)

Table 1: physmap characteristics across different architectures (x86, x86-64, AArch32, AArch64).

Initial Plan

- 1. Allocate a large number of ping socket objects and then free all of them by triggering the bug.
- 2. Iteratively call `mmap()` in the user program and fill the area.
- Hope the memory collision will happen?
 - Low success rate
 - Never let all the freed targeted vulnerable objects gather at one place



Reliability of Re-filling

Universal Solution #2

Reliable Memory Collision

- Goal: To make space for Physmap filled with our payload collide with PING sock objects in kernel
- Spray PING socket objects
 - In each step, every 500 **PADDING** PING objects
 - normally release them by close()
 - 1 **TARGET** PING objects
 - Used to pwn and trigger the bug to release them
- That makes our vulnerable PING socket objects appear everywhere in kernel space
 - Scatter anywhere
 - As long as one among these are overwritten, we win!

Information Leakage

Universal Solution #3


```
60 int inet_ioctl(struct socket *sock,
61               unsigned int cmd, unsigned long arg)
62 {
63     struct sock *sk = sock->sk;
64     int err = 0;
65     struct net *net = sock_net(sk);
66
67     switch (cmd) {
68     case SIOCGSTAMP:
69         err = sock_get_timestamp(sk,
70                                 (struct timeval __user *)arg);
71         break;
72     case SIOCGSTAMPNS:
73         err = sock_get_timestampns(sk,
74                                   (struct timespec __user *)arg);
75         break;
76     [...]
```

```
79 int sock_get_timestampns(struct sock *sk,
80                          struct timespec __user *userstamp)
81 {
82     struct timespec ts;
83     if (!sock_flag(sk, SOCK_TIMESTAMP))
84         sock_enable_timestamp(sk, SOCK_TIMESTAMP);
85     ts = ktime_to_timespec(sk->sk_stamp);
86     if (ts.tv_sec == -1)
87         return -ENOENT;
88     if (ts.tv_sec == 0) {
89         sk->sk_stamp = ktime_get_real();
90         ts = ktime_to_timespec(sk->sk_stamp);
91     }
92     return copy_to_user(userstamp, &ts,
93                        sizeof(ts)) ? -EFAULT : 0;
94 }
95 EXPORT_SYMBOL(sock_get_timestampns);
```

The data at a certain offset inside the object can be achieved by **ioctl()**

Find an info leak to know whether our targeting PING socket object has already been covered by physmap or not

Notice: certain adjustment and optimization in practical root tool

- ◉ Allocate hundreds of PING socket objects in group.
 - ◉ Every 500 padding objects with 1 targeting object considered as a vulnerable one.
- ◉ Free padding PING socket objects normally by calling `close()`
- ◉ Free targeting PING socket objects by triggering the bug
 - ◉ Such de-allocation generates large pieces of free memory (prepared for physmap)
- ◉ Iteratively call `mmap()` in user space and fill the areas
 - ◉ Payload + magic number for re-filling checking
- ◉ Iteratively call `ioctl()` on targeting PING socket objects
 - ◉ `ioctl()` returns magic number? Done.
 - ◉ Otherwise further physmap spraying is needed.

Summary

UNLEASH KERNEL UAF

- By leveraging physmap, we overcome all the challenges when exploiting such a UAF bug in kernel
- In fact, it is a generic memory collision model in Linux kernel
- Hard to mitigate due to kernel's inherent property

PC CONTROL

- Now we have full control of the content of a freed PING object with the corresponding dangling fd in our hand
- In user space: simply `close(fd)`
- In kernel:
 - `inet_release()` invoked
 - `sk_prot` is overwritten to a prepared virtual address in user space
 - Then PC value is under our control

```

97  int inet_release(struct socket *sock)
98  {
99      struct sock *sk = sock->sk;
100
101      if (sk) {
102          long timeout;
103
104          [...]
105
106          if (sock_flag(sk, SOCK_LINGER) &&
107              !(current->flags & PF_EXITING))
108              timeout = sk->sk_lingertime;
109          sock->sk = NULL;
110          sk->sk_prot->close(sk, timeout);
111      }
112      return 0;
113  }
114  EXPORT_SYMBOL(inet_release);

```


What does ShellCode do

- ◉ Leak kernel **sp** value (stack address)
 - ◉ Thus we get address of **thread_info**
- ◉ Overwrite **addr_limit** of the current thread to 0x0
- ◉ Then we achieve **kernel arbitrary read/write through pipe**

What about 64bit devices?

- Bug existed? **Yes.**
- LIST_POISON2?
 - Still 0x200200 which can be mapped. **Yes.**
- Memory collision with phsymap? **Yes.**
- Return to shellcode in user space? **No.**

Bypassing PXN

- PXN prevents userland execution from kernel
- Return to **physmap**? **Not executable** on phones ;(
- ROP comes on stage
 - **First step**: leak kernel stack address
 - **Second step**: change `addr_limit` to 0
- Hardcoded addresses of gadgets ;(

In fact, we perform JOP

- ◉ Avoid stack pivoting in kernel which brings uncertainty
- ◉ Make full use of current values of the registers
 - ◉ X29 stores SP value on 64bit devices
- ◉ High 32bits of kernel addresses are the same
 - ◉ Only need to read/write low 32bits
- ◉ Work hard to find cool gadgets

Conclusion

- We successfully root most popular Android devices on market.
- Android version ≥ 4.3
- First 64bit root case in the world (Samsung S6)
- Warranty safe

Roadmap

- CVE-2015-3636
- Kernel Exploit
- Future

64bit Devices could Be More Secure

- ◉ LIST_POISON2 in 64bit Android kernel
 - ◉ `0x200200` Set as `0xDEAD000000000000`
- ◉ Prevent memory collision with `physmap`
 - ◉ To impose restrictions on memory resources for every user
- ◉ KASLR
- ◉ Days become harder for linux kernel pwners
 - ◉ Where there is a will there is a way

ACKNOWLEDGEMENT

- Thanks for contributions and inspirations
 - Wushi
 - JFang
 - Leo.C
 - Liang Chen
 - Slipper
 - Peter



Thank you!
Q & A